SOFTWARE EVOLUTION AND THE FAULT PROCESS

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@ jpl.nasa.gov

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson @cs.uidaho.edu

ABSTRACT

In developing a software system, we would like to estimate the way in which the fault content changes during its development, as well determine the locations having the highest concentration of faults. In the phases prior to test, however, there may be very little direct information regarding the number and location of faults. This lack of direct information requires developing a fault surrogate from which the number of faults and their location can be estimated. We develop a fault surrogate based on changes in the fault index, a synthetic measure which has been successfully used as a fault surrogate in previous work. We show that changes in the fault index can be used to estimate the rates at which faults are inserted into a system between successive revisions. We can then continuously monitor the total number of faults inserted into a system, the residual fault content, and identify those portions of a system requiring the application of additional fault detection and removal resources.

1. INTRODUCTION

Over a number of years of study, we can now establish a distinct relationship between software faults and certain aspects of software complexity. When a software system consisting of many distinct software modules is built for the first time, we have little or no direct information as to the location of faults in the code. Some of the modules will have far more faults in them then do others. We do, however, now know that the number of faults in a module is highly correlated with certain software attributes that may be measured. This means that we can measure the software on these specific attributes and have some reasonable notion as to the degree to which the modules are fault prone [Muns90, Muns96].

In the absence of information as to the specific location of software faults, we have successfully used a derived metric, the fault index measure, as a fault surrogate. That is, if the fault index of a module is large, then it will likely have a large number of latent faults. If, on the other hand, the fault index of a module is small, then it will tend to have fewer faults. As the software system evolves through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code, however, is a fault prone process just as was the initial code generation. Faults may well be injected during this evolutionary process.

Code does not always change just to fix faults that have been isolated in it. Some changes to code during its evolution represent enhancements, design modifications or changes in the code in response to continually evolving requirements. These incremental code enhancements may also result in the introduction of still more faults. Thus, as a system progresses through a series of builds, the fault index of each program module that has been altered must also change. We will see that the rate of change in the system fault index will serve as a good index of the rate of fault introduction.

The general notion of software test is to make the rate of fault removal exceed the rate of fault introduction. In most cases, this is probably true [Muns97]. Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise. We would be very mistaken, then, to assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault introduction. The rate of fault removal is relatively easy to measure. The rate of fault introduction is much more tenuous. This fault introduction process is directly related to two measures that we can take on code as it evolves, fault deltas and net fault change (NFC).

In this investigation we establish a methodology whereby code can be measured from one build to the next, a measurement baseline. We use this measurement baseline to develop an assessment of the rate of change to a system as measured by our fault. From this change process we are then able to derive a direct measure of the rate of fault introduction based on changes in the software from one build to the next. Finally we examine data from an actual system on which faults may be traced to specific build increments to assess the predicted rate of fault introduction with the actual.

A major objective of this study is to identify a complete software system on which every version of every module has been archived together with the faults that have been recorded against the system as it evolved. For our purposes, the Cassini Orbiter Command and Data Subsystem at JPL met all of our objectives. On the first build of this system there were approximately 96K source lines of code in approximately 750 program modules. On the last build there were approximately 110K lines of source code in approximately 800 program modules. As the system progressed from the first to the last build there were a total of 45,200 different versions of these modules. On the average, then, each module progressed through an average of 60 evolutionary steps or versions. For the purposes of this study, the Ada program module is a procedure or function. it is the smallest unit of the Ada language structure that may be measured. A number of modules present in the first build of the system were removed on subsequent builds. Similarly, a number of modules were added.

The Cassini CDS does not represent an extraordinary software system. It is quite typical of the amount of change activity that will occur in the development of a system on the order of 100 KLOC. It is a non-trivial measurement problem to track the system as it evolves. Again, there are two different sets of measurement activities that must occur at once. We are interested the changes in the source code and we are interested in the fault reports that are being filed against each module.

2. A MEASUREMENT BASELINE

The measurement of an evolving software system through the shifting sands of time is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. This problem is very similar to that encountered by the surveying profession. If we were to buy a piece of property, there are certain physical attributes that we would like to know about that property. Among these properties is the topology of the site. To establish the topological characteristics of the land, we will have to seek out a benchmark. This benchmark represents an arbitrary point somewhere on the subject property. The distance and the elevation of every other point on the property may then be established in relation to the measurement baseline. Interestingly enough, we can pick any point on the property, establish a new baseline, and get exactly the same topology for the property. The property does not change. Only our perspective changes.

When measuring software evolution, we need to establish a measurement baseline for this same purpose [Niko97, Muns96a]. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges, only the perspective changes. The individual points involved in measuring software evolution are individual builds of the system.

For each raw metric in the baseline build, we may compute a mean and a standard deviation. Denote the vector of mean values for the baseline build as $\bar{\mathbf{x}}^B$ and the vector of standard deviations as $\mathbf{s}^B$. The standardized baseline metric values for any module j in an arbitrary build i, then, may be derived from raw metric values as

$$z_j^{B,i} = \frac{w_j^{B,i} - \bar{x}_j^B}{s_j^B}$$

Standardizing the raw metrics makes them more tractable. It now permits the comparison of metric values from one build to the next. From a software engineering perspective, there are simply too many metrics collected on each module over many builds. We need to reduce the dimensionality of the problem. We have successfully used principal components analysis for reducing the dimensionality of the problem [Muns90a, Khos92]. The principal components technique will reduce a set of highly correlated metrics to a much smaller set of uncorrelated or orthogonal measures. One of the products of the principal components technique is an orthogonal transformation matrix **T** that will send the standardized scores (the matrix **z**) onto a reduced set of domain scores thusly, $\mathbf{d} = \mathbf{zT}$.

In the same manner as the baseline means and standard deviations were used to transform the raw metric of any build relative to a baseline build, the transformation matrix $\mathbf{T}^B$ derived from the baseline build will be used in subsequent builds to transform standardized metric values obtained from that build to the reduced set of domain metrics as follows: $\mathbf{d}^{B,i} = \mathbf{z}^{B,i}\,\mathbf{T}^B$, where $\mathbf{z}^{B,i}$ are the standardized metric values from build i baselined on build B.

Another artifact of the principal components analysis is the set of eigenvalues that are generated for each of the new principal components. Associated with each of the new measurement domains is an eigenvalue, λ. These eigenvalues are large or small varying directly with the proportion of variance explained by each principal component. We have successfully exploited these eigenvalues to create the fault index, ρ, that is the weighted sum of the domain metrics to wit: $\rho_i = 50 + 10\sum_{j=1}^{m} \lambda_j d_j$, where m is the dimensionality of the reduced metric set [Muns90a].

As was the case for the standardized metrics and the domain metrics, the fault index may be baselined as well, using the eigenvalues and the baselined domain values:

$$\rho_i^B = \sum_{j=1}^{m} \lambda_j^B d_j^B$$

If the raw metrics that are used to construct the fault index are carefully chosen for their relationship to software faults then the fault index will vary in exactly the same manner as the faults [Muns95]. The fault index is a very reliable fault surrogate. Whereas we cannot measure the faults in a program directly we can measure the fault index of the program modules that contain the faults. Those modules having a large fault index will ultimately be found to be those with the largest number of faults [Muns92].

3. SOFTWARE EVOLUTION

A software system consists of one or more software modules. As the system grows and modifications are made, the code is recompiled and a new version, or build, is created. Each build is constructed from a set of software modules. The new version may contain some of the same modules as the previous version, some entirely new modules and it may even omit some modules that were present in an earlier version. Of the modules that are common to both the old and new version, some may have undergone modification since the last build. When evaluating the change that occurs to the system between any two builds (software evolution), we are interested in three sets of modules. The first set, M_C, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, M_A, is the set of modules that were in the early build and were removed prior to the later build. The final set, M_B, is the set of modules that have been added to the system since the earlier build.

The fault index of the system R^i at build i, the early build, is given by

$$R^i = \sum_{c \in M_c} \rho_c^i + \sum_{a \in M_a} \rho_a^i .$$

Similarly, the fault index of the system R^j at build j, the later build is given by

$$R^j = \sum_{c \in M_c} \rho_c^j + \sum_{b \in M_b} \rho_b^j .$$

The later system build is said to be more fault prone if $R_j > R_i$.

As a system evolves through a series of builds, its fault burden will change. This burden may be estimated by a set of software metrics. One simple assessment of the size of a software system is the number of lines of code per module. However, using only one metric may neglect information about the other complexity attributes of the system, such as control flow and temporal complexity. By comparing successive builds on their domain metrics it is possible to see how these builds either increase or decrease based on particular attribute domains. Using the fault index, the overall system fault burden can be monitored as the system evolves.

Regardless of which metric is chosen, the goal is the same. We wish to assess how the system has changed, over time, with respect to that particular measurement. The concept of a code delta provides this information. A code delta is, as the name implies, the difference between two builds as to the relative complexity metric.

The change in the fault in a single module between two builds may be measured in one of two distinct ways. First, we may simply compute the simple difference in the module fault index between build i and build j. We have called this value the fault delta for the module m, or $\delta_m^{i,j} = \rho_m^j - \rho_m^i$. A limitation of measuring fault deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the fault delta for the system will be close to zero. The overall complexity of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely affected by the replacing old modules with new ones. What we need is a measure to accompany fault delta that indicates how much change has occurred.

The absolute value of the fault delta is a measure of code churn. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault insertion, removing a lot of code is probably as catastrophic as adding a bunch. The new measure of net fault change (NFC), χ, for module m is simply

$$\chi_m^{i,j} = \left|\delta_m^{i,j}\right| = \left|\rho_m^i - \rho_{mJ}\right|$$

The total change of the system is the sum of the fault delta's for a system between two builds i and j is given by

$$\Delta^{i,j} = \sum_{c \in M_c} \delta_c^{i,j} - \sum_{a \in M_a} \rho_a^i + \sum_{b \in M_b} \rho_b^j .$$

Similarly, the NFC of the same system over the same builds is

$$\nabla^{i,j} = \sum_{c \in M_c} \chi_c^{i,j} + \sum_{a \in M_a} \rho_a^i + \sum_{b \in M_b} \rho_b^j .$$

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by a selected metric, fault delta, or we can measure the total amount of change the system has undergone between builds, net fault change.

4. OBTAINING AVERAGE BUILD VALUES

One synthetic software measure, fault index, has clearly been established as a successful surrogate measure of software faults [Muns90a]. It seems only reasonable that we should use it as the measure against which we compare different builds. Since the fault index is a composite measure based on the raw measurements, it incorporates the information represented by *LOC*, *V(g)*, η_1, η_2, and all the other raw metrics of interest. The fault index is a single value that is representative of the complexity of the system which incorporates all of the software attributes we have measured (e.g. size, control flow, style, data structures, etc.).

By definition, the average fault index, $\bar{\rho}$, of the baseline *system* will be

$$\bar{\rho}^B = \frac{1}{N^B}\sum_{i=1}^{N^B}\rho_i^B = 50,$$

where N^B is the cardinality of the set of modules on build *B*, the baseline build. The fault index for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The fault indices are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average fault index for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The average fault index for subsequent builds is given by

$$\bar{\rho}^k = \frac{1}{N^k}\sum_{i=1}^{N^k}\rho_i^{B,k},$$

where N^k is the cardinality of the set of program modules in the k^{th} build and $\rho_i^{B,k}$ is the baselined fault index for the i^{th} module of that set.

As the code is modified over time, faults will be found and fixed. However, new faults will be introduced into the code as a result of the change. In fact, this fault introduction process is directly proportional to change in the program modules from one version to the next. As a module is changed from one build to the next in response to evolving requirements changes and fault reports, its measurable software attributes will also change. Generally, the net effect of a change is that complexity will increase. Only rarely will its complexity decrease.

5. DEFINITION OF A FAULT

Unfortunately there is no particular definition of precisely what a software fault is. This makes it difficult to develop meaningful associative models between faults and metrics. In calibrating our model, we would like to know how to count faults in an accurate and repeatable manner. In measuring the evolution of the system to talk about rates of fault introduction and removal, we measure in units to the way that the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. Since the measurements of system structure are collected at the module level (by module we mean procedures and functions), we would like information about faults at the same granularity. We would also like to know if there are quantities that are related to fault counts that can be used to make our calibration task easier.

Following the second definition of fault in [IEEE83, IEEE88], we consider a fault to be a **structural imperfection** in a software system that **may** lead to the system's eventually failing. In other words, it is a **physical characteristic** of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. Faults are introduced into a system by people making errors in their tasks - these errors may be errors of commission or errors of omission. In order to count faults, we needed to develop a method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our structural measurements. Faults may be local – for instance, a system might contain an implementation fault affecting only one module in which the programmer incorrectly initializes a variable local to the routine. Faults may also span multiple modules - for instance, each module containing an include file with a particular fault would have that fault. In identifying and counting faults, we must deal with both types of faults. Details of the fault counting and identification rules developed for this study are given in [Niko97a, Niko98]

In analyzing the flight software for the CASSINI project the fault data and the source code change data were available from two different systems. The problem reporting information was obtained from the JPL institutional problem reporting system. Failures were recorded in this system starting at subsystem-level integration, and continuing through spacecraft integration and test. Failure reports typically contain descriptions of the failure at varying levels of detail, as well as descriptions of what was done to correct the fault(s) that caused the failure. Detailed information regarding the underlying faults (e.g., where were the code changes made in each affected module) is generally unavailable from the problem reporting system.

The entire source code evolution history could be obtained directly from the Software Configuration Control System (SCCS) files for all versions of the flight software. The way in which SCCS was used in this development effort makes it possible to track changes to the system at a module level in that each SCCS file stores the baseline version of that file (which may contain one or more modules) as well as the changes required to produce each subsequent increment (SCCS delta) of that file. When a module was created, or changed in response to a failure report or engineering change request, the file in which the module is contained was checked into SCCS as a new delta. This allowed us to track changes to the system at the module level as it evolved over time. For approximately 10% of the failure reports, we were able to identify the source file increment in which the fault(s) associated with a particular failure report were repaired. This information was available either in the comments inserted by the developer into the SCCS file as part of the check-in process, or as part of the set of comments at the beginning of a module that track its development history.

Using the information described above, we performed the following steps to identify faults. First, for each problem report, we searched all of the SCCS files to identify all modules and the increment(s) of each module for which the software was changed in response to the problem report. Second, for each increment of each module identified in the previous step, we assumed as a starting point that all differences between the increment in which repairs are implemented and the previous increment are due solely to fault repair. Note that this is not necessarily a valid assumption - developers may be making functional enhancements to the system in the same increment that fault repairs are being made. Careful analysis of failure reports for which there was sufficiently detailed descriptive information served to separate areas of fault repair from other changes. However, the level of detail required to perform this analysis was not consistently available. Third, we used a differential comparator (e.g., Unix `diff`) to obtain the differences between the increment(s) in which the fault(s) were repaired, and the immediately preceding increment(s). The results indicated the areas to be searched for faults.

After completing the last step, we still had to identify and count the faults - the results of the differential comparison cannot simply be counted up to give a total number of faults. In order to do this, we developed a taxonomy for identifying and counting faults [Niko98]. This taxonomy differs from others in that it does not seek to identify the root cause of the fault. Rather, it is based on the types of changes made to the software to repair the faults associated with failure reports - in other words, it constitutes an operational definition of a fault. Although identifying the root causes of faults is important in improving the development process [Chil92, IEEE93], it is first necessary to identify the faults. We do not claim that this is the only way to identify and count faults, nor do we claim that this taxonomy is complete. However, we found that this taxonomy allowed us to successfully identify faults in the software used in the study in a consistent manner at the appropriate level of granularity.

6. THE RELATIONSHIP BETWEEN FAULTS AND CODE CHANGES

Having established a theoretical relationship between software faults and code changes, it is now of interest to validate this model empirically. This measurement occurred on two simultaneous fronts. First, all of the versions of all of the source code modules were measured. From these measurements, NFC and fault deltas were obtained for every version of every module. The failure reports were sampled to lead to specific faults in the code. These faults were classified according to the above taxonomy manually on a case by case basis. Then we were able to build a regression model relating the code measures to the code faults.

The Ada source code modules for all versions of each of these modules were systematically reconstructed from the SCCS code deltas. Each of these module versions was then measured by the UX-Metric analysis tool for Ada [SETL93]. Not all metrics provided by this tool were used in this study. Only a subset of these actually provide distinct sources of variation [Khos90]. The specific metrics used in this study are shown in Table 1.

Metrics	Definition
η_1	Count of unique operators [Hal77]
η_2	Count of unique operands
N_1	Count of total operators
N_2	Count of total operands
P/R	Purity ratio: ratio of Halstead's $\hat{N}$ to total program vocabulary
V(g)	McCabe's cyclomatic complexity
Depth	Maximum nesting level of program blocks
AveDepth	Average nesting level of program blocks
LOC	Number of lines of code
Blk	Number of blank lines
Cmt	Count of comments
CmtWds	Total words used in all comments
Stmts	Count of executable statements
LSS	Number of logical source statements
PSS	Number of physical source statements
NonEx	Number of non-executable statements
AveSpan	Average number of lines of code between references to each variable
Vl	Average variable name length

Table 1. Software Metric Definitions

To establish a baseline system, all of the metric data for the module versions that were members of the first build of CDS were then analyzed by our PCA-FI tool. This tool is designed to compute fault indices either from a baseline system or from a system being compared to the baseline system. In that the first build of the Cassini CDS system was selected to be the baseline system, the PCA-FI tool performed a principal components analysis on these data with an orthogonal varimax rotation. The objective of this phase of the analysis is to use the principal components technique to reduce the dimensionality of the metric set. As may been seen in Table 2, there are four principal components for the 18 metrics shown in Table 1. For convenience, we have chosen to name these principal components as **Size**, **Structure**, **Style** and **Nesting**. From the last row in Table 2 we can see that the new reduced set of orthogonal components of the original 18 metrics account for approximately 85% of the variation in the original metric set.

Metric	Size	Structure	Style	Nesting
Stmts	0.968	0.022	-0.079	0.021
LSS	0.961	0.025	-0.080	0.004
N_2	**0.926**	0.016	0.086	0.086
N_1	**0.934**	0.016	0.074	0.077
η_2	**0.884**	0.012	-0.244	0.043
AveSpan	**0.852**	0.032	0.031	-0.082
V(g)	**0.843**	0.032	-0.094	-0.114
η_1	**0.635**	-0.055	-0.522	-0.136
Depth	**0.617**	-0.022	-0.337	-0.379
LOC	-0.027	**0.979**	0.136	0.015
Cmt	-0.046	**0.970**	0.108	0.004
PSS	-0.043	**0.961**	0.149	0.019
CmtWds	0.033	**0.931**	0.058	-0.010
NonEx	-0.053	**0.928**	0.076	-0.009
Blk	0.263	**0.898**	0.048	0.005
P/R	-0.148	-0.198	**-0.878**	0.052
VI	0.372	-0.232	**-0.752**	0.010
AveDepth	-0.000	-0.009	0.041	**-0.938**
% Variance	37.956	30.315	10.454	6.009

Table 2. Principal Components of Software Metrics

As is typical in the principal components analysis of metric data, the **Size** domain dominates the analysis. It alone accounts for approximately 38% of the total variation in the original metric set. Not surprisingly, this domain contains the metrics of total statement count (*Stmts*), logical source statements (LSS), the Halstead lexical metric primitives of operator and operand count, but it also contains cyclomatic complexity (*V(g)*). In that we regularly find cyclomatic complexity in this domain we are forced to conclude that it is only a simple measure of size in the same manner as statement count. The **Structure** domain contain those metrics relating to the physical structure of the program such as non-executable statements (*NonEx*) and the program block count (*Blk*). The **Style** domain contains measures of attribute that are directly under a programmer's control such as variable length (*VI*) and purity ratio (*P/R*). The **Nesting** domain consist of the single metric that is a measure of the average depth of nesting of program modules (*AveDepth*).

In order to transform the raw metrics for each module version into their corresponding fault indices, the means and the standard deviations must be computed. These values will be used to transform all raw metric values for all versions of all modules to their baselined z score values. The transformation matrix will then map the metric z score values onto their orthogonal equivalents to obtain the orthogonal domain metric values used in the computation of the fault index. With this information, we can obtain baselined fault index values for any version of any module relative to the baseline build. As an aside, it is not necessary that the baseline build be the initial build. As a typical system progresses through hundreds of builds in the course of its life, it is worth reestablishing a baseline closer to the current system. In any event, these baseline data are saved by the PCA-FI tool for use in later computation of metric values. Whenever the tool is invoked referencing the baseline data it will automatically use these data to transform the raw metric values given to it.

Once the baselined fault index data have been assembled for all versions of all modules, it is then possible to examine some trends that have occurred during the evolution of the system. For example, in Figure 1 the fault index of the evolving CDS system is shown across one of its five major builds. To compute these changing fault index values, every development increment within that build was identified. Then, for each increment, the baselined fault indices of the modules in that increment were computed. The next four increments, not shown here, have evolutionary patterns similar to that shown in Figure 1. It seems to be that the average fault index of most systems is a monotonically increasing function.

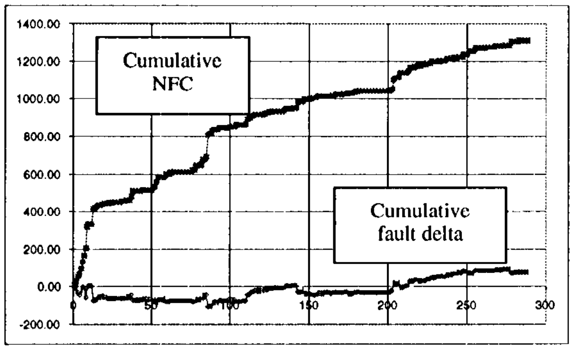

Figure 1. Change in the Fault Index for One Version of CDS Flight Software

Note in Figure 1 that not all increments within a build represent the same increase in the fault index. Nearly one third of the total change in this version takes place within the first 10% of the development increments. From our understanding of the relationship between the fault index and injected faults, we would expect that the magnitude of change within the first 30 increments would indicate that a large number of faults would have been injected as a result of this activity. It is also interesting to note that the final fault index of this particular version is rather close to the initial fault index, although it is quite clear from the measured activity that a significant amount of change has occurred.

Not all program modules received the same degree of modification as the system evolved. Some modules changed relatively little. Figure 2 shows the net fault change and fault delta values for a module that was relatively stable over its change history. There were only four relatively minor changes to this module. A more typical change history is shown for another module in Figure 3. The total net fault change for this module is approximately 38. It is interesting to note that the fault delta for this module is close to zero. The fault index of the module at the last version is very close to its original value. This figure clearly illustrates the conceptual differences between the two measure of net fault change and fault delta.

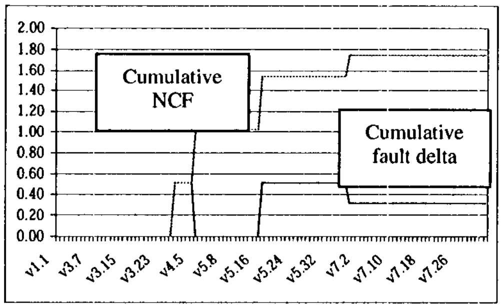

Figure 2. Change History for Stable Module

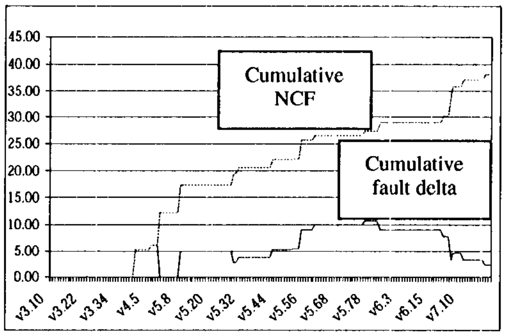

Figure 3. Typical Module Change History

Figure 4 shows a module at the extreme end of change history. This module has a total net fault change value of close to 140. Also, its final fault delta value is about 30, indicating that its fault index has also increased significantly as it evolved. Among the three modules whose change history is illustrated by Figures 2, 3, and 4, the latter module is the one that we focus our attention on the most. It is the one most likely to have had significant numbers of faults introduced into it throughout its dramatic life.

Now let us turn our attention to the fault identification process. Over 600 failure reports were written against the CDS flight software during developmental testing and system integration. Failure reports contain a description of how the system's behavior deviated from expectations, the date on which the failure was observed, and a description of the corrective action that was taken.

In relating the number of faults inserted in an increment to measures of a module's structural change, we had only a small number of observations with which to work. There were three difficulties that had to be dealt with. First, recall that for only about 10% of the failure reports were we able to identify the module(s) that had been changed, and in which increment those changes were made. Although the development practices used on this project included the placement of comments in the source code to identify repair activities resulting from each problem report, this requirement was not consistently enforced. Second, once a fault had been identified, it was necessary to trace it back to the increment in which it first occurred. For some source files, there were over 100 increments that had to be manually searched. Since the SCCS files for each delivered version were available, it was possible to trace most faults back to their point of origin. As previously noted, the principal difficulty was the sheer volume of material that had to be examined – this was one of the factors restricting the number of observations that could be obtained. Third, there were numerous instances in which the UX-Metric analyzer that was used to obtain the raw structural measurements would not measure a particular module. The net result was that of the over 100 faults that were initially identified, there were only 35 observations in which a fault could be associated with a particular increment of a module, and with that increment's measures of fault delta and net fault change.

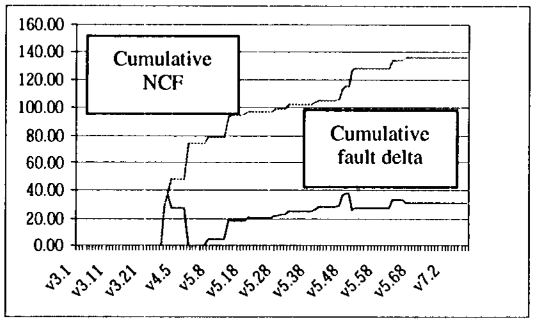

Figure 4. Change History for Frequently Changed Module

For each of the 35 modules for which there was viable fault data, there were three data points. First, we had the number of injected faults for that module that were the direct result of changes that had occurred on that module between the current version that contained the faults and the previous version that did not. Second, we had fault delta values for each of these modules from the current to the previous version. Finally, we had net fault change values derived from the fault deltas.

Linear regression models were computed for net fault change and fault deltas with actual code faults as the dependent variable in both cases. Both models were build without constant terms in that we surmise that if no changes were made to a module, then no new faults could be introduced. The results of the regression between faults and fault deltas were not at all surprising. The squared multiple R for this model was 0.001, about as close to zero as you can get. This result is directly attributable to the non-linearity of the data. Change comes in two flavors. Change may increase the complexity of a module. Change may decrease the complexity of a model. Faults, on the other hand are not related to the direction of the change but to its intensity. Removing masses of code from a module is just as likely to introduce faults and adding code to it.

The regression model between net fault change and faults is dramatically different. The regression ANOVA for this model are shown in Table 3. Whereas fault deltas do not show a linear relationship with faults, net fault change certainly does. The actual regression model is given in Table 4. In Table 5 the regressions statistics have been reported. Of particular interest is the Squared Multiple R term, having a value of 0.653. This means, roughly, that the regression model will account for more than 65% of the variation in the faults of the observed modules based on the values of net fault change.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	331.879	1	331.879	62.996	0.000
Residual	179.121	34	10.673	5.268	

Table 3. Regression Analysis of Variance

Effect	Coefficient	Std Err	t	P(2-Tail)
NFC	0.576	0.073	7.937	0.000

Table 4. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	0.806	0.649	2.296

Table 5. Regression Statistics

Of course, it may be the case that both the amount of change and the direction in which the change occurred. The linear regression through the origin shown in Tables 6, 7, and 8 below illustrates this model.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	367.247	2	183.623	42.153	0.000
Residual	143.753	33	4.356		

Table 6. Regression Analysis of Variance

Effect	Coefficient	Std Err	t	P(2-Tail)
NFC	0.647	0.071	9.172	0.000
Delta	0.201	0.071	2.849	0.002

Table 7. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	.848	.719	2.087

Table 8. Regression Statistics

We see that the model incorporating fault delta as well as net fault change performs significantly better than the model incorporating net fault change alone, as measured by Squared Multiple R and Mean Sum of Squares.

We determined whether the linear regression model which uses net fault change alone is an adequate predictor at a particular significance level when compared to the model using both net fault change and fault delta. We used the R^2-adequate test [MacD97, Net83] to examine the linear regression models through the origin and determine whether the models that depend only on structural measures are an adequate predictor. A subset of predictor variables is said to be R^2-adequate at significance level α if:

$R^2_{sub} > 1-(1-R^2_{full})(1+d_{n,k})$, where

- R^2_{sub} is the R^2 value achieved with the subset of predictors
- R^2_{full} is the R^2 value achieved with the full set of predictors
- $d_{n,k} = (kF_{k,n-k-1})/n-k-1$, where
  - k = number of predictor variables in the model
  - n = number of observations
  - F = F statistic for significance α for n,k degrees of freedom.

Table 9 below show values of R^2, k, degrees of freedom, $F_{k,n-k-1}$, $d_{n,k}$, and R^2_{sub} for all four linear regression models through the origin. The number of observations, n, is 35, and we specify a value of α=.05.

We see in Table 9 that the value of Multiple Squared R for the regression using only net fault change is 0.649, and the 5% significance threshold for the net fault change and fault delta regression model is 0.661. This means that the regression model using only NFC is not R^2 adequate when compared to the model using both net fault change and fault delta as predictors. The amount of change occurring between subsequent revisions and the direction of that change both appear to be important in determining the number of faults inserted into a system.

Lin. Regressions Through Origin	R^2	DF	k	$F_{k,n-k-1}$ for significance α	d(n,k)	Threshold for significance α
NFC only	0.649	34	1	4.139	0.125	-----
NFC, Fault Delta	0.719	33	2	3.295	0.206	0.661

Table 9. Values of R^2, DOF, k, $F_{k,n-k-1}$, and $d_{n,k}$ for R^2-adequate Test

Finally, we examined the predicted residuals for the linear regression models described above. Table 10 below shows the results of the Wilcoxon Signed Ranks test, as applied to the predictions for the excluded observations and the number of faults observed for each of the two linear regression models through the origin. For these models, about 2/3 of the estimates tend to be less than the number of faults observed.

Plots of the predicted residuals against the actual number of observed faults for each of the linear regression models through the origin are shown in Figures 5 and 6 below. The results of the Wilcoxon signed ranks tests, as well as Figures 5 and 6, indicate that the predictive accuracy of the regression models might be improved if syntactic analyzers capable of measuring additional aspects of a software system's structure were available. Recall, for instance, that we did not measure any of the real-time aspects of the system. Analyzers capable of measuring changes in variable definition and usage as well changes to the sequencing of blocks might also provide more accurate measurements.

Sample Pair		N	Mean Rank	Sum of Ranks	Test Statistic Z	Asymptotic Significance (2-tailed)
Observed Faults; NFC only fault est.	Neg.	25[a]	17.52	438.00	-2.015[d]	.044
	Pos.	10[b]	19.20	192.00		
	Ties	0[c]				
	Total	35				
Observed Faults; NFC and Fault Delta est.	Neg.	24[a]	16.92	406.00	-1.491[d]	.136
	Pos.	11[b]	20.36	224.00		
	Ties	0[c]				
	Total	35				

a. Observed Faults > Regression model predictions
b. Observed Faults < Regression model predictions
c. Observed Faults = Regression model predictions
d. Based on positive ranks

Table 10. Wilcoxon Signed Ranks Test for Linear Regressions Through the Origin

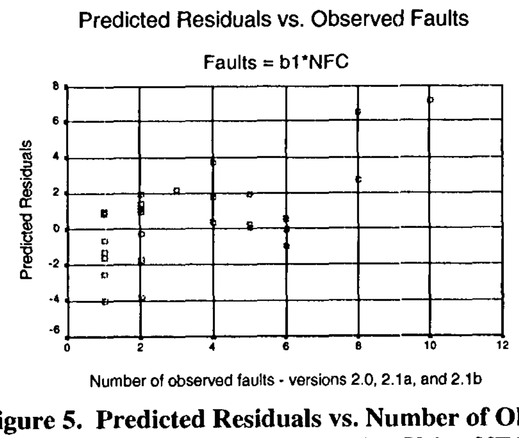

Figure 5. Predicted Residuals vs. Number of Observed Faults for Linear Regression Using NFC

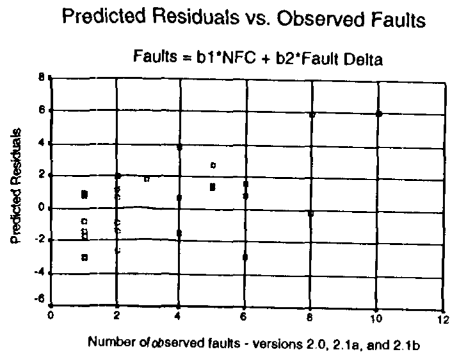

Figure 6. Predicted Residuals vs. Number of Observed Faults for Linear Regression with NFC and Fault Delta

7. SUMMARY

There is a distinct and a strong relationship between software faults and measurable software attributes. This is in itself not a new result or observation. The most interesting result of this endeavor is that we also found a strong association between the fault introduction process over the evolutionary history of a software system and the degree of change taking place in each of the program modules. We also found that the direction of the change was significant in determining the number of faults inserted. Some changes will have the potential of introducing very few faults while others may have a serious impact on the number of latent faults. Different numbers of faults may be inserted, depending upon whether code is being added to or removed from the system.

In order for the measurement process to be meaningful, fault data must be very carefully collected. In this study, the data were extracted ex post facto as a very labor intensive effort. Since fault data cannot be collected with the same degree of automation as much of the data on software metrics being gathered by development organizations, material changes in the software development and software maintenance processes must be made to capture these fault data. Among other things, a well defined fault standard and fault taxonomy must be developed and maintained as part of the software development process. Further, all designers and coders should be trained in its use. A viable standard is one that may be used to classify any fault unambiguously. A viable fault recording process is one in which any one person will classify a fault exactly the same as any other person.

Finally, the whole notion of measuring the fault introduction process is its ultimate value as a measure of software process. The software engineering literature is replete with examples of how software process improvement can be achieved through the use of some new software development technique. What is almost absent from the same literature is a controlled study to validate the fact that the new process is meaningful. The techniques developed in this study can be implemented in a development organization to provide a consistent method of measuring fault content and structural evolution across multiple projects over time. We are working with software development efforts at JPL to address the practical aspects of inserting these measurement techniques into production software development environments. The initial estimates of fault insertion rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault insertion rate, and to identify those development techniques that seem to provide the greatest reduction.

ACKNOWLEDGMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[Chil92] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurement", IEEE Transactions on Software Engineering, November, 1992, pp. 943-946.

[Hal77] M. H. Halstead, *Elements of Software Science.* Elsevier, New York, 1977.

[IEEE83] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[IEEE88] "IEEE Standard Dictionary of Measures to Produce Reliable Software", IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.

[IEEE93] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994

[Khos90] T. M. Khoshgoftaar and J. C. Munson , "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* 8, 1990, pp. 253-261.

[Khos92] T. M. Khoshgoftaar and J. C. Munson "A Measure of Software System Complexity and Its Relationship to Faults," In *Proceedings of the 1992 International Simulation Technology Conference*, The Society for Computer Simulation, San Diego, CA, 1992, pp. 267-272.

[MacD97] S. G. MacDonell, M. J. Shepperd, P. J. Sallis, "Metrics for Database Systems: An Empirical Study", Proceedings of the Fourth International Software Metrics Symposium, November 5-7, 1997, Albuquerque, NM, pp. 99-107

[Muns90] J. C. Munson and T. M. Khoshgoftaar "Regression Modeling of Software Quality: An Empirical Investigation," *Journal of Information and Software Technology*, 32, 1990, pp. 105-114.

[Muns90a] J. C. Munson and T. M. Khoshgoftaar "The Relative Software Complexity Metric: A Validation Study," In *Proceedings of the Software Engineering 1990 Conference*, Cambridge University Press, Cambridge, UK, 1990, pp. 89-102.

[Muns92] J. C. Munson and T. M. Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18, No. 5, 1992, pp. 423-433.

[Muns95] J. C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering,* J. C. Baltzer AG, Amsterdam 1995.

[Muns96] J. C. Munson, "Software Faults, Software Failures, and Software Reliability Modeling", *Information and Software Technology*, December, 1996.

[Muns96a] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium* , IEEE Computer Society Press, pp. 41-51.

[Muns97] J. C. Munson and G. A. Hall, "Estimating Test Effectiveness with Dynamic Complexity Measurement," *Empirical Software Engineering Journal.* Feb. 1997.

[Net83] J. Neter, W. Wasserman, M. H. Kutner, Applied Linear Regression Models, Irwin: Homewood, IL, 1983

[Niko97] A. P. Nikora, N. F. Schneidewind, J. C. Munson, "IV&V Issues in Achieving High Reliability and Safety in Critical Control System Software", proceedings of the International Society of Science and Applied Technology conference, March 10-12, 1997, Anaheim, CA, pp 25-30.

[Niko97a] A. P. Nikora, J. C. Munson, "Finding Fault with Faults: A Case Study", proceedings of the Annual Oregon Workshop on Software Metrics, Coeur d'Alene, ID, May 11-13, 1997

[Niko98] A. P. Nikora, "Software System Defect Content Prediction From Development Process And Product Characteristics", Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[SETL93] "User's Guide for UX-Metric 4.0 for Ada", SET Laboratories, Mulino, OR, © SET Laboratories, 1987-1993